

1. 初识Vue

新型框架 MVVM: Model-View-ViewModel (虚拟DOM节点、DIFF算法)

- Model: 模型, 你把数据处理好 (比如: Ajax请求到服务器拿数据)
- View: 视图, 你把页面布局好;
- ViewModel: 剩下的事交给我 (Vue), 由 (Vue) 来完成View和Model之间的数据渲染;

把页面结构 (节点结构) 放到一个对象里面 (虚拟节点: VNode), 当数据发生改变的时候, 会使用DIFF算法, 去对比你修改了什么, 能够继续使用的节点不修改, 尽可能少的修改页面节点 (修改数据和修改页面显示是异步的) 。

- 想让Vue工作, 就必须创建一个Vue实例, 且要传入一个配置对象
- demo容器里的代码依然符合html规范, 只不过混入了一些特殊的Vue语法
- demo容器里的代码被称为【Vue模板】
- Vue实例和容器是一一对应的
- 真实开发中只有一个Vue实例, 并且会配合着组件一起使用
- {{xxx}}是Vue的语法: 插值表达式, {{xxx}}可以读取到data中的所有属性
- 一旦data中的数据发生改变, 那么页面中用到该数据的地方也会自动更新(Vue实现的响应式)

vue3示例:

index.js

```
1  const app = createApp({
2    // data() 返回的属性将会成为响应式的状态
3    // 并且暴露在 `this` 上
4    data: function () { //使用函数来返回数据 (model): 形成一个封闭的空间
5      return {
6        message: '哈, 光哥你好',
7        count: 100,
8        list: []
9      };
10   },
11   // methods 是一些用来更改状态与触发更新的函数
12   // 它们可以在模板中作为事件监听器绑定
13   methods: { // 在这里写方法
14     bindEvent() {
15       this.message = 'guanf is good'
16     }
17   },
18   // 生命周期钩子会在组件生命周期的各个不同阶段被调用
19   // 例如这个函数就会在组件挂载完成后被调用
```

```

20     mounted() {
21         console.log(`The initial count is ${this.count}.`)
22     }
23 });
24 // 把app实例挂载到 app模版中
25 app.mount('#app'); // 应用实例必须在调用了 .mount() 方法后才会渲染出来

```

index.html

```

1 <div id="app">
2   <button v-on:click="bindEvent">点击</button>
3   <h1>
4     {{message}}
5   </h1>
6 </div>

```

多个应用实例:

createApp API 允许你在同一个页面中创建多个共存的 Vue 应用，而且每个应用都拥有自己的用于配置和全局资源的作用域。

```

1 const app1 = createApp({
2   /* ... */
3 })
4 app1.mount('#container-1')
5
6 const app2 = createApp({
7   /* ... */
8 })
9 app2.mount('#container-2')

```

Vue 的两个核心功能:

1. **声明式渲染**: Vue 基于标准 HTML 拓展了一套模板语法，使得我们可以声明式地描述最终输出的 HTML 和 JavaScript 状态之间的关系。
2. **响应性**: Vue 会自动跟踪 JavaScript 状态变化并在改变发生时响应式地更新 DOM。

2. 模版语法

2.1 文本插值

最基本的数据绑定形式，双大括号标签会被替换为相应组件实例中 msg 属性的值

- 插值语法:

- 功能：用于解析标签体内容
- 写法：{{xxx}}，xxx是js表达式，且可以直接读取到data中的所有属性

```
1 <span>Message: {{ msg }}</span>
```

2.2 原始HTML

双大括号将会将数据插值为纯文本，而不是 HTML。若想插入 HTML，你需要使用 v-html 指令：

- 指令语法：
- 功能：用于解析标签（包括：标签属性、标签体内容、绑定事件.....）
- 举例：v-bind:href="xxx" 或 简写为 :href="xxx"，xxx同样要写js表达式，且可以直接读取到data中的所有属性

```
1 <p>
2   {{ rawHtml }}
3 </p>
4
5 <p>
6   <span v-html="rawHtml"></span>
7 </p>
```

v-html attribute 被称为一个指令。指令由 v- 作为前缀，表明它们是一些由 Vue 提供的特殊 attribute。

安全警告：

在网站上动态渲染任意 HTML 是非常危险的，因为这很容易造成 XSS 漏洞。请仅在内容安全可信时再使用 v-html，并且永远不要使用用户提供的 HTML 内容。

3. 数据绑定

单向数据绑定 (v-bind)：数据只能从data流向页面

v-bind 指令指示 Vue 将元素的 id attribute 与组件的 demo 属性保持一致。如果绑定的值是 null 或者 undefined，那么该 attribute 将会从渲染的元素上移除。

双向绑定(v-model)：数据不仅能从data流向页面，还可以从页面流向data

提示：

- 1.双向绑定一般都应用在表单类元素上（如：input、select等）
- 2.v-model:value 可以简写为 v-model，因为v-model默认收集的就是value值

eg:

```
1 <!--
```

```

2    单向数据绑定: v-bind 数据从data流向页面
3
4    -->
5    <div id="root">
6        单向数据绑定: <input type="text" v-bind:value="name">
7        <br>
8        双向数据绑定: <input type="text" v-model:value="name">
9
10       <!-- 简写 -->
11       <!-- 单向数据绑定: <input type="text" :value="name"> -->
12       <!-- 双向数据绑定: <input type="text" v-model="name"> -->
13       <br>
14
15       <!-- v-model:
16           只能应用在表单类, 输入类元素上
17       -->
18       <!-- <h2 v-model:value="name">hello world!!!</h2> -->
19   </div>

```

动态绑定多个值:

```

1  data() {
2    return {
3      demo: {
4        id: 'container',
5        class: 'wrapper'
6      }
7    }
8  }
9
10  <div v-bind="demo"></div>
11

```

使用 JavaScript 表达式:

```

1  {{ number + 1 }}
2
3  {{ ok ? 'YES' : 'NO' }}
4

```

```

5  {{ message.split('').reverse().join('') }}
6
7  <div :id="`list-${id}`"></div>

```

在 Vue 模板内，JavaScript 表达式可以被使用在如下场景上：

- 在文本插值中 (双大括号)
- 在任何 Vue 指令 (以 v- 开头的特殊 attribute) attribute 的值中

调用函数：

```

1  <span :title="toTo(date)">
2    {{ formDt(date) }}
3  </span>

```

参数 Arguments:

```

1  <a v-bind:href="url"> ... </a>
2
3  <!-- 简写 -->
4  <a :href="url"> ... </a>
5
6  <a v-on:click="doSomething"> ... </a>
7
8  <!-- 简写 -->
9  <a @click="doSomething"> ... </a>
10

```

4. 响应式基础

4.1 声明响应式状态

对象的所有顶层属性都会被代理到组件实例

```

1  export default {
2    data() {
3      return {
4        count: 1,
5        itemArray:[] // 若所需的值还未准备好，在必要时也可以使用 null、undefined 或者其他一些
                      // 值占位。
6      }
7    },

```

```

8
9 // `mounted` 是生命周期钩子，之后我们会讲到
10 mounted() {
11   // `this` 指向当前组件实例
12   console.log(this.count) // => 1
13
14   // 数据属性也可以被更改
15   this.count = 2
16 }
17 }

```

4.2 声明方法

要为组件添加方法，我们需要用到 `methods`（对象）。

```

1 export default {
2   data() {
3     return {
4       count: 0
5     }
6   },
7   methods: { // 可能要写多个方法，因此methods为对象
8     increment() {
9       this.count++
10    },
11    // 错误的方法
12    sum(x,y)=>{
13      // 无法访问此处的 `this`！
14      return x + y;
15    }
16  },
17  mounted() {
18    // 在其他方法或是生命周期中也可以调用方法
19    this.increment()
20  }
21 }

```

`methods` 中的方法绑定指向组件实例的 `this`。确保了方法在作为事件监听器或回调函数时始终保持正确的 `this`。 **不应该在定义 `methods` 时使用箭头函数**，因为箭头函数没有自己的 `this` 上下文。

```
1 <button @click="increment">{{ count }}</button>
```

和组件实例上的其他属性一样，方法也可以在模板上被访问。在模板中它们常常被用作事件监听器。

DOM 更新时机:

可以使用 `nextTick()` 这个全局 API 等待一个状态改变后的 DOM 更新完成。

```
1 import { nextTick } from 'vue'
2
3 export default {
4   methods: {
5     increment() {
6       this.count++
7       nextTick(() => {
8         // 访问更新后的 DOM
9       })
10    }
11  }
12 }
```

深层响应性:

在vue中，状态都是默认深层响应式的。这意味着在更高层次的对象或数组，有改动就能检测到。

有状态方法:

应该在created生命周期钩子中创建函数，例如防抖函数

5、计算属性（用来描述依赖响应式状态的复杂逻辑）

- 定义：要用的属性不存在，要通过已有属性计算得来
- 原理：底层借助了Object.defineProperty方法提供的getter和setter
- get函数什么时候执行？
 - (1).初次读取时会执行一次
 - (2).当依赖的数据发生改变时会被再次调用
- 优势：与methods实现相比，内部有缓存机制（复用），效率更高，调试方便
- 备注：
 - 计算属性最终会出现在vm上，直接读取使用即可
 - 如果计算属性要被修改，那必须写set函数去响应修改，且set中要引起计算时依赖的数据发生改变

index.HTML



```
1 <div id="app">
2   <div @click="countFn">
3     计数器: {{count}}
4   </div>
5   <div>
6     {{desk}}张桌子共有{{leg}}条腿
7   </div>
8   <button type="button" @click="buyDesk">买一张桌子</button>
9   <button type="button" @click="buyLegFour">买4个桌子腿</button>
10 </div>
```



```

1  import { createApp } from 'vue';
2
3
4  const app = createApp({
5    data: function () {
6      return {
7        desk: 3,
8        count: 0
9      };
10   },
11   methods: {
12     buyDesk() {
13       ++this.desk;
14     },
15     countFn() {
16       this.count++;
17     },
18     legNum() {
19       console.log('methos', '桌子腿');
20       return this.desk * 4;
21     },
22     buyLegFour() {
23       //会自动调用计算属性的leg的set方法
24       this.leg += 4;
25     }
26   },
27   computed: {
28     leg: {
29       //正常来讲就是get
30       get() {
31         console.log('computed', '桌子腿');
32         return this.desk * 4;
33       },
34       //我不能根据统计出来的商品数量及总价反向推导出每个商品的价格
35       //我不能根据学生的平均成绩反向推到出学生的数量及每个学生的成
36       绩
37       set(newValue) {
38         this.desk = newValue / 4;
39       }
40     }
41   },
42 });
43 app.mount('#app');

```

一个计算属性仅会在其响应式依赖更新时才重新计算。这意味着只要 不改变，无论多少次访问 而不用重复执行 getter 函数。

计算属性缓存 vs 方法：

```

1  <p>{{ fn() }}</p>
2
3  // 组件中

```

```

4 methods: {
5   fn() {
6     return this.author.books.length > 0 ? 'Yes' : 'No'
7   }
8 }

```

将同样的函数定义为一个方法而不是计算属性，两种方式在结果上确实是完全相同的，然而，不同之处在于计算属性值会基于其响应式依赖被缓存。一个计算属性仅会在其响应式依赖更新时才重新计算。这意味着只要 `author.books` 不改变，无论多少次访问 **fn函数** 都会立即返回先前的计算结果，而不用重复执行 getter 函数。

计算函数不应有副作用：

计算属性的 计算函数应该只做计算而没有其他的副作用。例如：不要在计算函数中做异步请求或者更改DOM。

6、类与样式绑定

数据绑定的常见场景是操作元素的CSS class列表和内联样式，可以使用v-bind将他们与动态的字符串绑定。

6.1、绑定HTML class

给 :class (v-bind:class 的缩写) 传递一个对象来动态切换 class

```
<div :class="{ active: isActive }"></div>
```

active 是否存在取决于数据属性 isActive 的真假值，我们可以在对象中写多个字段来操作多个 class，:class指令可以与一般的class属性共存。

```

1 data() {
2   return {
3     isActive: true,
4     hasError: false
5   }
6 }
7 // 模版
8 <div
9   class="static"
10  :class="{ active: isActive, 'text-danger': hasError }"
11 >
12 </div>
13
14 // 渲染结果

```

```
15 <div class="static active"></div>
```

绑定的对象并不一定需要写成字面量的形式，可以直接绑定一个对象。当然也可以绑定一个返回对象的计算属性。

```
1 data() {
2   return {
3     isActive: true,
4     error: null,
5     classObject1: {
6       active: true,
7       'text-danger': false
8     }
9   }
10 },
11 computed: {
12   classObject2() {
13     return {
14       active: this.isActive && !this.error,
15       'text-danger': this.error && this.error.type === 'fatal'
16     }
17   }
18 }
19 <div :class="classObject1"></div>
20 <div :class="classObject2"></div>
```

6.2、绑定数组

给 :class 绑定一个数组来渲染多个 CSS class:

```
1 data() {
2   return {
3     activeClass: 'active',
4     errorClass: 'text-danger'
5   }
6 }
7 <div :class="[activeClass, errorClass]"></div>
8 // 渲染的结果是:
9 <div class="active text-danger"></div>
```

有条件地渲染某个 class, 可以使用三元表达式:

```
1 <div :class="[isActive ? activeClass : '', errorClass]"></div>
```

在数组中嵌套对象:

```
1 <div :class="[ { active: isActive }, errorClass]"></div>
```

6.3、在组件上使用

对于只有一个根元素的组件, 当你使用了 class attribute 时, 这些 class 会被添加到根元素上, 并与该元素上已有的 class 合并。

```
1 <!-- 子组件模板 -->
```

```
2 <p class="foo bar">Hi!</p>
```

```
3
```

```
4 <!-- 在使用组件时 -->
```

```
5 <MyComponent class="baz boo" />
```

```
6
```

```
7 渲染出的 HTML 为:
```

```
8 <p class="foo bar baz boo">Hi</p>
```

```
9
```

```
10 Class 的绑定也是同样的:
```

```
11 <MyComponent :class="{ active: isActive }" />
```

```
12
```

```
13 当 isActive 为真时, 被渲染的 HTML 会是:
```

```
14 <p class="foo bar active">Hi</p>
```

```
15
```

```
16 组件有多个根元素, 需要指定根元素来接收这个 class, 可以通过组件的 $attrs 属性来实现指定:
```

```
17 <!-- MyComponent 模板使用 $attrs 时 -->
```

```
18 <p :class="$attrs.class">Hi!</p>
```

```
19 <span>This is a child component</span>
```

```
20
```

```
21 <MyComponent class="baz" />
```

```
22
```

```
23
```

```
24 渲染为:
```

```
25 <p class="baz">Hi!</p>
26 <span>This is a child component</span>
```

6.4、绑定内联样式

绑定对象：

`:style` 支持绑定 `JavaScript` 对象值，对应的是 HTML 元素的 `style` 属性：

```
1 data() {
2   return {
3     activeColor: 'red',
4     fontSize: 30
5   }
6 }
7
8 <div :style="{ color: activeColor, fontSize: fontSize + 'px' }"></div>
9
```

直接绑定一个样式对象通常是一个好主意，这样可以使模板更加简洁：

```
1 data() {
2   return {
3     styleObject: {
4       color: 'red',
5       fontSize: '13px'
6     }
7   }
8 }
9
10 <div :style="styleObject"></div>
```

绑定数组：

给 `:style` 绑定一个包含多个样式对象的数组。这些对象会被合并后渲染到同一元素上：

```
1 <div :style="[baseStyles, overridingStyles]"></div>
```

自动前缀：

当在 `:style` 中使用了需要浏览器特殊前缀的 CSS 属性时，Vue 会自动为他们加上相应的前缀。

样式多值：一个样式属性提供多个 (不同前缀的) 值，

```
1 <div :style="{ display: ['-webkit-box', '-ms-flexbox', 'flex'] }"></div>
```

7、条件渲染

v-if: v-if 指令用于条件性地渲染一块内容。这块内容只会在指令的表达式返回真值时才被渲染

- 写法:
 - (1).v-if="表达式"
 - (2).v-else-if="表达式"
 - (3).v-else="表达式"
- 适用于：切换频率较低的场景
- 特点：不展示的DOM元素直接被移除
- 注意：v-if可以和v-else-if、v-else一起使用，但要求结构不能被“打断”

<template> 上的 v-if:

v-if 是一个指令，他必须依附于某个元素，如果我们想要切换不止一个元素，可以在一个<template> 元素上使用 v-if，这只是一个不可见的包装器元素，最后渲染的结果并不会包含这个<template> 元素。

```
1 <template v-if="ok">
2   <h1>Title</h1>
3   <p>Paragraph 1</p>
4   <p>Paragraph 2</p>
5 </template>
```

v-show:

- 写法：v-show="表达式"
- 适用于：切换频率较高的场景
- 特点：不展示的DOM元素未被移除，仅仅是使用样式隐藏掉(display:none); v-show 不支持在<template> 元素上使用，也不能和 v-else 搭配使用。

```
1 <div id="app">
2   <!-- v-if -->
3   <button @click="showBox">盒子显示与隐藏</button>
4   <div class="box" v-if="box">
5     i am hezi
6   </div>
7   <div class="nohappy" v-else>
```

```

8         Oh no 😞
9     </div>
10
11     <!-- v-show -->
12     <div class="vshow" v-show="box">
13         v-show
14     </div>
15
16     <!-- v-if v-else v-else-if -->
17     <div class="person">
18         <div class="left">
19             <div class="menu" v-on:click="showPersonSet(1)">个人设置</div>
20             <div class="menu" v-on:click="showPersonSet(2)">头像设置</div>
21             <div class="menu" v-on:click="showPersonSet(3)">密码修改</div>
22             <div class="menu" v-on:click="showPersonSet(4)">个性化设置</div>
23         </div>
24
25         <div class="right">
26             <div class="set" v-if="num == 1">
27                 个人设置 100
28             </div>
29             <div class="avatar" v-else-if="num == 2">
30                 头像修改 200
31             </div>
32             <div class="passwd" v-else-if="num == 3">
33                 密码修改 300
34             </div>
35             <div class="gexing" v-else>
36                 个性化设置 400
37             </div>
38         </div>
39
40     </div>
41
42     <template v-if="many"> // template不会在页面呈现
43         <div>
44             1
45         </div>
46         <div>
47             2

```

```
48         </div>
49         <h1>guang</h1>
50     </template>
51
52 </div>
```

8. 列表渲染

v-for指令

- 用于展示列表数据
- 语法: v-for="(item, index) in xxx" :key="yyy"
- 可遍历: 数组、对象、字符串 (用的很少)、指定次数 (用的很少)

使用 v-for 指令基于一个数组来渲染一个列表。v-for 指令的值需要使用 item in items 形式的特殊语法, 其中 items 是源数据的数组, 而 item 是迭代项的别名:

```
1 <div id="root">
2     <!-- 遍历数组 -->
3     <h2>人员列表 (遍历数组) </h2>
4     <ul>
5         <li v-for="(p,index) of persons" :key="index">
6             {{p.name}}-{{p.age}}
7         </li>
8     </ul>
9
10    <!-- 遍历对象 -->
11    <h2>汽车信息 (遍历对象) </h2>
12    <ul>
13        <li v-for="(value,k) of car" :key="k">
14            {{k}}-{{value}}
15        </li>
16    </ul>
17
18    <!-- 遍历字符串 -->
19    <h2>测试遍历字符串 (用得少) </h2>
20    <ul>
21        <li v-for="(char,index) of str" :key="index">
22            {{char}}-{{index}}
23        </li>
```



```

24     </ul>
25
26     <!-- 遍历指定次数 -->
27     <h2>测试遍历指定次数（用得少）</h2>
28     <ul>
29         <li v-for="(number,index) of 5" :key="index">
30             {{index}}-{{number}}
31         </li>
32     </ul>
33 </div>

```

index.ts

```

1  import { createApp } from 'vue';
2
3  const app = createApp({
4      data() {
5          return {
6              persons: [
7                  { id: '001', name: '张三', age: 18 },
8                  { id: '002', name: '李四', age: 19 },
9                  { id: '003', name: '王五', age: 20 }
10             ],
11             car: {
12                 name: '奥迪A8',
13                 price: '70万',
14                 color: '黑色'
15             },
16             str: 'hello'
17         }
18     },
19 });
20 app.mount('#app');

```

对于多层嵌套的v-for，作用域的工作方式和函数的作用域很类似，



```
<li v-for="item in items">
```

```
    <span v-for="childItem in item.children">
```

```
{{ item.message }} {{ childItem }}  
  
</span>  
  
</li>
```

可以使用 `of` 作为分隔符来替代 `in`，这更接近 JavaScript 的迭代器语法：

```
😊 <div v-for="item of items"> </div>
```

v-for域对象

使用 `v-for` 来遍历一个对象的所有属性。遍历的顺序会基于对该对象调用 `Object.keys()` 的返回值来决定。

```
😊 data() {  
  return {  
    myObject: {  
      title: 'How to do lists in Vue',  
      author: 'Jane Doe',  
      publishedAt: '2016-04-10'  
    }  
  }  
}  
  
模版：  
  
<ul>  
  <li v-for="value in myObject">  
    {{ value }}  
  </li>  
</ul>
```

可以通过提供第二个参数表示属性名 (例如 `key`):

```
1 <li v-for="(value, key) in myObject">
2   {{ key }}: {{ value }}
3 </li>
4 第三个参数表示位置索引:
5 <li v-for="(value, key, index) in myObject">
6   {{ index }}. {{ key }}: {{ value }}
7 </li>
```

同时使用 v-if 和 v-for 是不推荐的，因为这样二者的优先级不明显。

为了给 Vue 一个提示，以便它可以跟踪每个节点的标识，从而重用和重新排序现有的元素，你需要为每个元素对应的块提供一个唯一的 key attribute：

```
😊 <div v-for="item in items" :key="item.id">
  <!-- 内容 -->
</div>
```

当你使用 <template v-for> 时，key 应该被放置在这个 <template> 容器上：

```
😊 <template v-for="todo in todos" :key="todo.name">
  <li>{{ todo.name }}</li>
</template>
```

9、事件处理

监听事件

😊 使用 v-on 指令 (简写为 @) 来监听 DOM 事件，并在事件触发时执行对应的 JavaScript。用法：v-on:click="methodName" 或 @click="handler"。

事件处理器的值可以是：

1. 内联事件处理器：事件被触发时执行的内联 JavaScript 语句 (与 onclick 类似)。
2. 方法事件处理器：一个指向组件上定义的方法的属性名或是路径。

😊 内联事件处理器：

```
data() {
  return {
```

```
      count: 0
    }
  }

模版:

<button @click="count++">Add 1</button>

<p>Count is: {{ count }}</p>
```

😄 方法事件处理器：随着事件处理器的逻辑变得愈发复杂，内联代码方式变得不够灵活。因此 `v-on` 也可以接受一个方法名或对某个方法的调用。

```
data() {
  return {
    name: 'Vue.js'
  }
},
methods: {
  greet(event) {
    // 方法中的 `this` 指向当前活跃的组件实例
    alert(`Hello ${this.name}!`)

    // `event` 是 DOM 原生事件
    if (event) {
      alert(event.target.tagName)
    }
  }
}

<!-- greet 是上面定义过的方法名 -->
```

```
<button @click="greet">Greet</button>
```

方法事件处理器会自动接收原生 DOM 事件并触发执行。在上面的例子中，我们能够通过被触发事件的 `event.target.tagName` 访问到该 DOM 元素。



在内联处理器中调用方法:

除了直接绑定方法名，你还可以在内联事件处理器中调用方法。这允许我们向方法传入自定义参数以代替原生事件：

```
methods: {
```

```
  say(message) {
```

```
    alert(message)
```

```
  }
```

```
}
```

```
<button @click="say('hello')">Say hello</button>
```

```
<button @click="say('bye')">Say bye</button>
```



在内联事件处理器中访问事件参数:

有时我们需要在内联事件处理器中访问原生 DOM 事件。你可以向该处理器方法传入一个特殊的 `$event` 变量，或者使用内联箭头函数：

```
<!-- 使用特殊的 $event 变量 -->
```

```
<button @click="warn('Form cannot be submitted yet.', $event)">
```

```
  Submit
```

```
</button>
```

```
<!-- 使用内联箭头函数 -->
```

```
<button @click="(event) => warn('Form cannot be submitted yet.', event)">
```

Submit

</button>

```
methods: {
```

```
  warn(message, event) {
```

```
    // 这里可以访问 DOM 原生事件
```

```
    if (event) {
```

```
      event.preventDefault()
```

```
    }
```

```
    alert(message)
```

```
  }
```

```
}
```

事件修饰符

😄 在处理事件时调用 `event.preventDefault()` 或 `event.stopPropagation()` 是很常见的。尽管我们可以直接在方法内调用，但如果方法能更专注于数据逻辑而不用去处理 DOM 事件的细节会更好。为解决这一问题，Vue 为 `v-on` 提供了**事件修饰符**。修饰符是用 `.` 表示的指令后缀，包含以下这些：

- `.stop`
- `.prevent`
- `.self`
- `.capture`
- `.once`
- `.passive`

<!-- 单击事件将停止传递 -->

<a @click.stop="doThis">

<!-- 提交事件将不再重新加载页面 -->

<form @submit.prevent="onSubmit"> </form>

<!-- 修饰语可以使用链式书写 -->

```
<a @click.stop.prevent="doThat"> </a>
```

<!-- 也可以只有修饰符 -->

```
<form @submit.prevent> </form>
```

<!-- 仅当 event.target 是元素本身时才会触发事件处理器 -->

<!-- 例如：事件处理器不来自子元素 -->

```
<div @click.self="doThat">...</div>
```

使用修饰符时需要注意调用顺序，因为相关代码是以相同的顺序生成的。因此使用

`@click.prevent.self` 会阻止元素及其子元素的所有点击事件的默认行为而

`@click.self.prevent` 则只会阻止对元素本身的点击事件的默认行为。

`.capture`、`.once` 和 `.passive` 修饰符与原生 `addEventListener` 事件相对应：

<!-- 添加事件监听器时，使用 `capture` 捕获模式 -->

<!-- 例如：指向内部元素的事件，在被内部元素处理前，先被外部处理 -->

```
<div @click.capture="doThis">...</div>
```

<!-- 点击事件最多被触发一次 -->

```
<a @click.once="doThis"> </a>
```

<!-- 滚动事件的默认行为 (scrolling) 将立即发生而非等待 `onScroll` 完成 -->

<!-- 以防其中包含 `event.preventDefault()` -->

```
<div @scroll.passive="onScroll">...</div>
```

`.passive` 修饰符一般用于触摸事件的监听器，可以用来改善移动端设备的滚屏性能。

按键修饰符



Vue 允许在 `v-on` 或 `@` 监听按键事件时添加按键修饰符。

<!-- 仅在 key 为 Enter 时调用 submit -->

```
<input @keyup.enter="submit" />
```

使用 `KeyboardEvent.key` 暴露的按键名称作为修饰符，但需要转为 `kebab-case` 形式。

```
<input @keyup.page-down="onPageDown" />
```

按键别名:

Vue 为一些常用的按键提供了别名:

- `.enter`
 - `.tab`
 - `.delete` (捕获 “Delete” 和 “Backspace” 两个按键)
 - `.esc`
 - `.space`
 - `.up`
 - `.down`
 - `.left`
 - `.right`
-

系统按键修饰符#

使用以下系统按键修饰符来触发鼠标或键盘事件监听器，只有当按键被按下时才会触发。

- `.ctrl`
- `.alt`
- `.shift`
- `.meta`

```
<!-- Alt + Enter -->
```

```
<input @keyup.alt.enter="clear" />
```

```
<!-- Ctrl + 点击 -->
```

```
<div @click.ctrl="doSomething">Do something</div>
```

`.exact` 修饰符:

`.exact` 修饰符允许控制触发一个事件所需的确定组合的系统按键修饰符。

```
<!-- 当按下 Ctrl 时，即使同时按下 Alt 或 Shift 也会触发 -->
```



```
<button @click.ctrl="onClick">A</button>
```

<!-- 仅当按下 Ctrl 且未按任何其他键时才会触发 -->

```
<button @click.ctrl.exact="onCtrlClick">A</button>
```

<!-- 仅当没有按下任何系统按键时触发 -->

```
<button @click.exact="onClick">A</button>
```

鼠标按键修饰符:

- .left
- .right
- .middle

这些修饰符将处理程序限定为由特定鼠标按键触发的事件。

10、表单输入绑定

😊 `v-model` 指令帮我们简化了这一步骤:

```
<input v-model="text">
```

`v-model` 还可以用于各种不同类型的输入, `<textarea>`、`<select>` 元素。它会根据所使用的元素自动使用对应的 DOM 属性和事件组合:

- 文本类型的 `<input>` 和 `<textarea>` 元素会绑定 `value` property 并侦听 `input` 事件;
- `<input type="checkbox">` 和 `<input type="radio">` 会绑定 `checked` property 并侦听 `change` 事件;
- `<select>` 会绑定 `value` property 并侦听 `change` 事件:

基本用法



文本:

```
<p>Message is: {{ message }}</p>
```

```
<input v-model="message" placeholder="edit me" />
```

多行文本:

```
<span>Multiline message is:</span>
```

```
<p style="white-space: pre-line;">{{ message }}</p>
```

```
<textarea v-model="message" placeholder="add multiple lines"></textarea>
```

注意在 `<textarea>` 中是不支持插值表达式的。请使用 `v-model` 来替代：

```
<!-- 错误 -->
```

```
<textarea>{{ text }}</textarea>
```

```
<!-- 正确 -->
```

```
<textarea v-model="text"></textarea>
```

复选框：

单一的复选框，绑定布尔类型值：

```
<input type="checkbox" id="checkbox" v-model="checked" />
```

```
<label for="checkbox">{{ checked }}</label>
```



单选按钮

```
<div>Picked: {{ picked }}</div>
```

```
<input type="radio" id="one" value="One" v-model="picked" />
```

```
<label for="one">One</label>
```

```
<input type="radio" id="two" value="Two" v-model="picked" />
```

```
<label for="two">Two</label>
```

选择器

```
<div>Selected: {{ selected }}</div>
```

```
<select v-model="selected">
```

```
<option disabled value="">Please select one</option>
```

```
<option>A</option>
```

```
<option>B</option>
```

```
<option>C</option>
```

```
</select>
```

选择器的选项可以使用 v-for 动态渲染:

```
export default {
```

```
  data() {
```

```
    return {
```

```
      selected: 'A',
```

```
      options: [
```

```
        { text: 'One', value: 'A' },
```

```
        { text: 'Two', value: 'B' },
```

```
        { text: 'Three', value: 'C' }
```

```
      ]
```

```
    }
```

```
  }
```

```
}
```

```
<select v-model="selected">
```

```
<option v-for="option in options" :value="option.value">
```

```
  {{ option.text }}
```

```
</option>
```

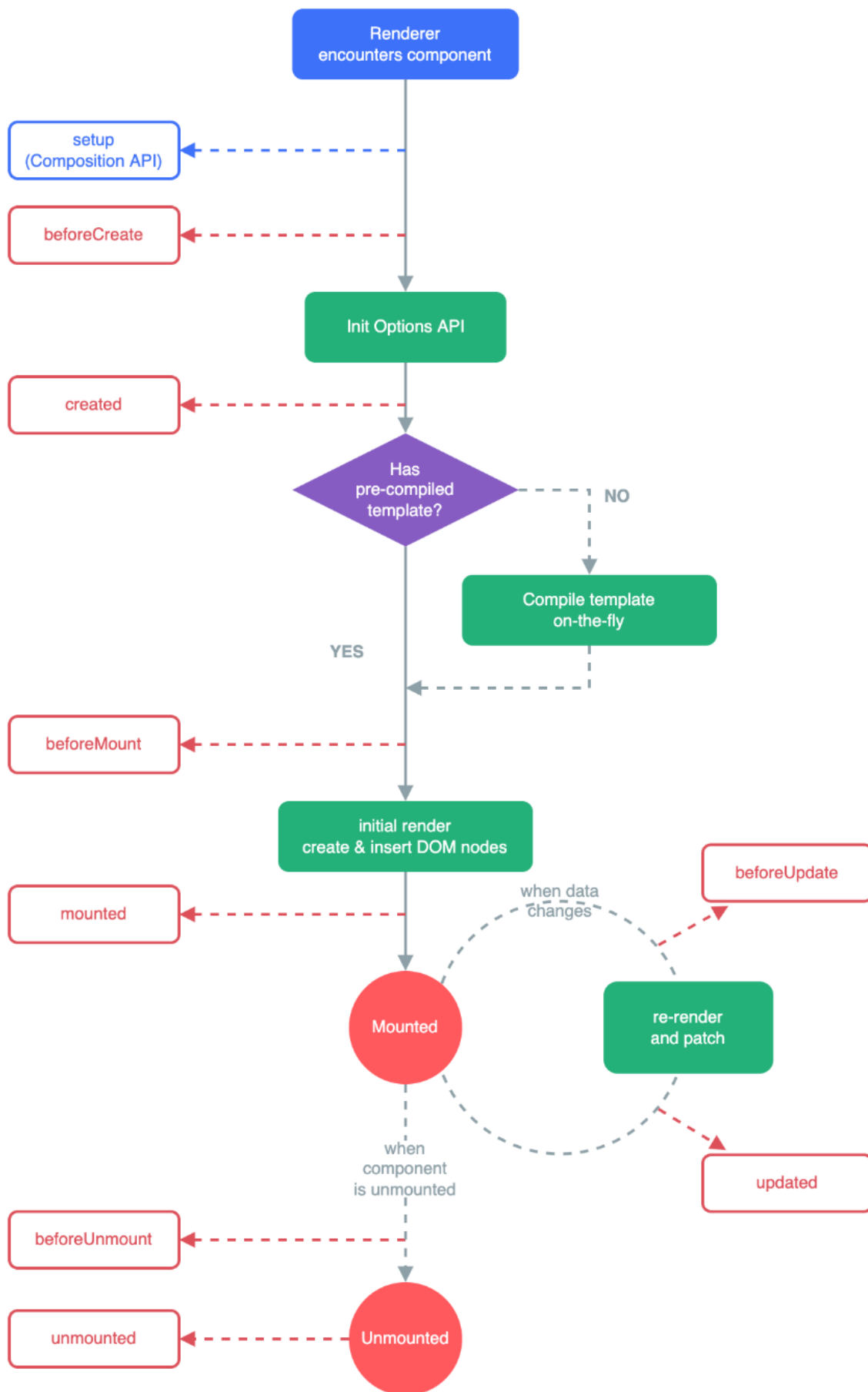
```
</select>
```

```
<div>Selected: {{ selected }}</div>
```

11、生命周期

所有生命周期钩子函数的 `this` 上下文都会自动指向当前调用它的组件实例。注意：避免用箭头函数来定义生命周期钩子，因为如果这样的话你将无法在函数中通过 `this` 获取组件实例。

生命周期图示



有关所有生命周期钩子及其各自用例的详细信息，请参考[生命周期钩子 API 索引](#)。



beforeCreate

在组件实例初始化完成之后立即调用。

会在实例初始化完成、props 解析之后、data() 和 computed 等选项处理之前立即调用。

注意，组合式 API 中的 setup() 钩子会在所有选项式 API 钩子之前调用，beforeCreate() 也不例外。



created

在组件实例处理完所有与状态相关的选项后调用。

当这个钩子被调用时，以下内容已经设置完成：响应式数据、计算属性、方法和侦听器。然而，此时挂载阶段还未开始，因此

\$el 属性仍不可用。



beforeMount

当这个钩子被调用时，组件已经完成了其响应式状态的设置，但还没有创建 DOM 节点。它即将首次执行 DOM 渲染过程。

这个钩子在服务端渲染时不会被调用。



mounted

在组件被挂载之后调用。

组件在以下情况下被视为已挂载：

- 所有同步子组件都已经被挂载。（不包含异步组件或<Suspense> 树内的组件）
- 其自身的 DOM 树已经创建完成并插入了父容器中。注意仅当根容器在文档中时，才可以保证组件 DOM 树也在文档中。

这个钩子通常用于执行需要访问组件所渲染的 DOM 树相关的副作用，或是在[服务端渲染应用](#)中用于确保 DOM 相关代码仅在客户端被调用。

这个钩子在服务端渲染时不会被调用。



beforeUpdate

在组件即将因为一个响应式状态变更而更新其 DOM 树之前调用。

这个钩子可以用来在 Vue 更新 DOM 之前访问 DOM 状态。在这个钩子中更改状态也是安全的。

这个钩子在服务端渲染时不会被调用。



updated

在组件因为一个响应式状态变更而更新其 DOM 树之后调用。

父组件的更新钩子将在其子组件的更新钩子之后调用。

这个钩子会在组件的任意 DOM 更新后被调用，这些更新可能是由不同的状态变更导致的。如果你需要在某个特定的状态更改后访问更新后的 DOM，请使用 [nextTick\(\)](#) 作为替代。

这个钩子在服务端渲染时不会被调用。



beforeUnmount

在一个组件实例被卸载之前调用。

当这个钩子被调用时，组件实例依然还保有全部的功能。

这个钩子在服务端渲染时不会被调用。



unmounted

在一个组件实例被卸载之后调用。

一个组件在以下情况下被视为已卸载：

- 其所有子组件都已经被卸载。
- 所有相关的响应式作用 (渲染作用以及 `setup()` 时创建的计算属性和侦听器) 都已经停止。

可以在这个钩子中手动清理一些副作用，例如计时器、DOM 事件监听器或者与服务器的连接。

这个钩子在服务端渲染时不会被调用。

12、侦听器

😄 计算属性允许我们声明性地计算衍生值。然而在有些情况下，我们需要在状态变化时执行一些“副作用”：例如更改 DOM，或是根据异步操作的结果去修改另一处的状态。

在选项式 API 中，我们可以使用 `watch` 选项在每次响应式属性发生变化时触发一个函数。

```
1 <!-- 监听 -->
2 <div id="app">
3   <input type="text" v-model="keywords" @keyup.enter="searchFn">
4 </div>
5
6 import { createApp } from "vue";
7
8 const app = createApp({
9   data() {
10     return {
11       keywords: '',
12       router: {
13         name: 'home',
14         compnent: 'App',
15         passwd: 'dd'
16       }
17     };
18   },
19   methods: {
20     searchFn() {
21       console.log('执行查询操作');
22     },
```



```

23     getConcatKeywords<T>(keywords: T) {
24         // 向服务器发起ajax请求，获取关联的数据信息
25         console.log(keywords);
26     }
27 },
28 // 1 侦听器 watch 选项在每次响应式属性发生变化是触发一个函数，做定向监视
29 watch: {
30     keywords(newValue, oldValue) {
31         console.log(newValue, oldValue);
32         this.getConcatKeywords(newValue)
33     },
34     // 2 watch 选项也支持把键设置成用 . 分隔的路径：
35     // 注意：只能是简单的路径，不支持表达式。
36     "some.nested.key"(newValue){
37         //...
38     }
39     // 3 针对属性进行深度监听
40     'router.name'() {
41         console.log(this.router.name);
42     },
43     // 4 深层监听
44     router: {
45         handler(newValue, oldValue) {
46             console.log(this);
47             console.log(newValue, oldValue);
48             console.log('你只改变了属性值.....');
49         },
50         deep: true,
51         immediate: true, // watch 默认是懒执行的：仅当数据源变化时，才会执行回
    调。立即执行
52         // 回调的触发时机：默认情况下,侦听器回调中访问的 DOM 将是被 Vue 更新之前
    的状态
53         flush: 'post' // 如果想在侦听器回调中能访问被 Vue 更新之后的DOM，你需
    要指明 flush: 'post' 选项：
54     }
55 }
56 });
57
58 app.mount("#app");

```

😄 `this.$watch()`

使用组件实例的 `$watch()` 方法来命令式地创建一个侦听器：

```
export default {  
  created() {  
    this.$watch('question', (newQuestion) => {  
      // ...  
    })  
  }  
}
```

运用场景：如果要在特定条件下设置一个侦听器，或者只侦听响应用户交互的内容，这方法很有用。它还允许你提前停止该侦听器。

停止侦听器：

用 `watch` 选项或者 `$watch()` 实例方法声明的侦听器，会在宿主组件卸载时自动停止。

在少数情况下，你的确需要在组件卸载之前就停止一个侦听器，这时可以调用

`$watch()` API 返回的函数：

```
const unwatch = this.$watch('foo', callback)  
// ...当该侦听器不再需要时  
unwatch()
```

13、模版引用



1、ref

某些情况下，我们人需要访问DOM元素，可以使用ref属性：

```
<input ref="input">
```

ref是一个特殊的属性，和v-for中的key类似，允许我们在一个特定的 DOM 元素或子组件实例被**挂载后**，获得对它的直接引用。运用场景：组件挂载时将焦点设置到一个 input 元素上，或在一个元素上初始化一个第三方库。

2、访问模板引用

挂载结束后，引用都会暴露在**this.\$refs**之上：

```
<script>
export default {
  mounted() {
    this.$refs.input.focus()
  }
}
</script>
<template>
<input ref="input" />
</template>
```

注意，你只可以在**在组件挂载后**才能访问模板引用，初次渲染时会为null。

🍷 3、v-for 中的模板引用：

当在 v-for 中使用模板引用时，相应的引用中包含的值是一个数组：

```
<script>
export default {
  data() {
    return {
      list: [
        /* ... */
      ]
    },
  },
  mounted() {
    console.log(this.$refs.items)
  }
}
</script>
<template>
<ul>
<li v-for="item in list" ref="items">
  {{ item }}
</li>
```

```
</ul>
```

```
</template>
```

应该注意的是，`ref` 数组并不保证与源数组相同的顺序。

4、函数模板引用

`ref` 除了使用字符串做名字外，还可以绑定一个函数，在组件更新时调用，该函数会收到元素引用作为其第一个参数：

```
<input :ref="(el) => { /* 将 el 赋值给一个数据属性或 ref 变量 */ }">
```

注意：要使用动态的 `:ref` 绑定才能传入一个函数，当绑定的元素被卸载时，函数也会被调用一次，此时的 `el` 参数会是 `null`。

5、组件上的 `ref`

模板引用也可以被用在一个子组件上。这种情况下引用中获得的值的是组件实例：

```
<script>
import Child from './Child.vue'
export default {
  components: {
    Child
  },
  mounted() {
    // this.$refs.child 是 <Child /> 组件的实例
  }
}
</script>
<template>
<Child ref="child" />
</template>
```

`expose` 选项可以用于限制对子组件实例的访问：

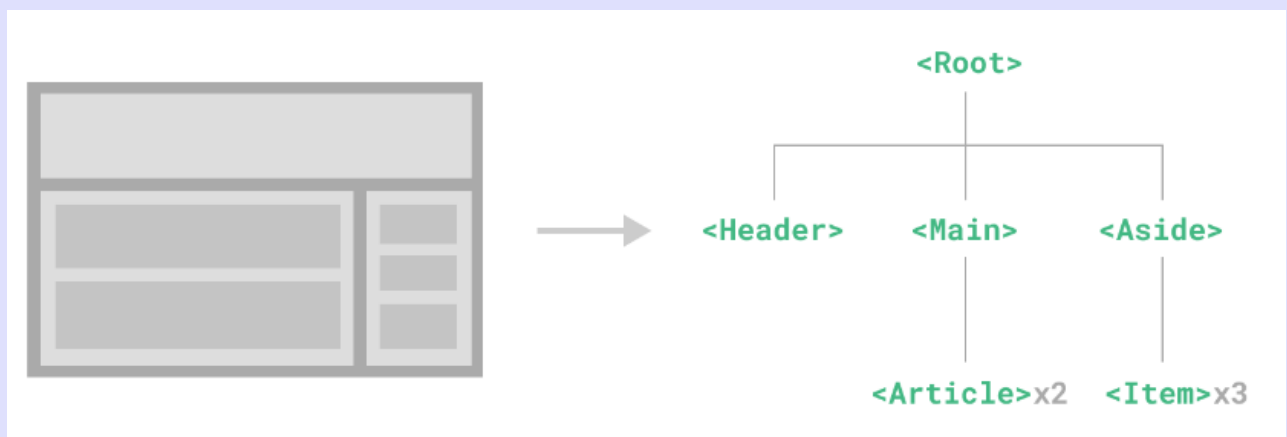
```
export default {
  expose: ['publicData', 'publicMethod'],
  data() {
    return {
      publicData: 'foo',
      privateData: 'bar'
    }
  }
}
```

```
}  
},  
methods: {  
  publicMethod() {  
    /* ... */  
  },  
  privateMethod() {  
    / ... */  
  }  
}
```

在上面这个例子中，父组件通过模板引用访问到子组件实例后，仅能访问 `publicData` 和 `publicMethod`。

14、组件基础

😊 组件允许我们将 UI 划分为独立的、可重用的部分，并且可以对每个部分进行单独的思考。在实际应用中，组件常常被组织成层层嵌套的树状结构：



1、定义一个组件：

当使用构建步骤时，我们一般会将vue组件定义在一个单独的.vue文件中，即为单文件组件（SEC）

使用组件：

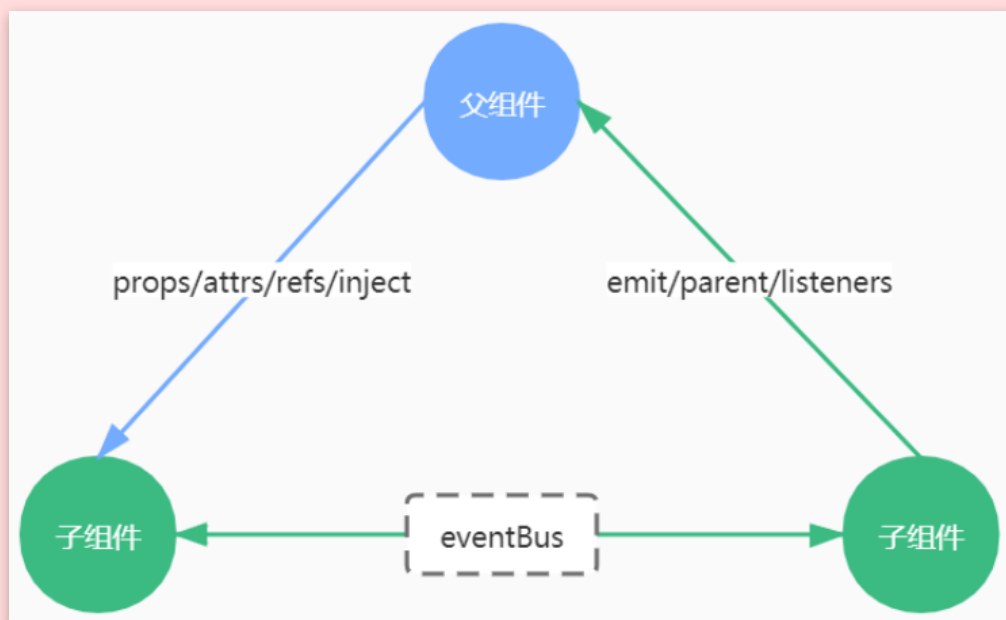
要使用一个组件时，我们需要从父组件中导入它

步骤：1、导入组件 2、注册组件

```
1 <template>
2   <my-nav> </my-nav>
3   <div class="container">
4     <ArticleList :cid="1" cate="科技" />
5     <ArticleList :cid="2" cate="创意" />
6     <ArticleList :cid="3" cate="文艺" />
7     <ArticleList :cid="4" cate="商业" />
8   </div>
9
10 </template>
11
12 <script lang="ts">
13   // 引入要使用的组件
14   import ArticleList from './component/ArticleList.vue';
15
16   export default {
17     data() {
18       return {};
19     },
20     methods: {},
21     // 组件的注册：局部
22     components: {
23       ArticleList
24     },
25   }
26 </script>
27 <style>
28   /* 全局的样式 */
29   @import '@/sass/index.scss';
30 </style>
31 <style lang="scss" scoped>
32   /* 当前组件要用的私有的样式 */
33   .mydiv {
34     background-color: #f01;
35   }
```



2、组件传值：



(1) 父组件传给子组件

在子组件里定义一个props,即props:['msg'], msg可以是对象也可以是基本数据类型

如果你想定义一个默认值, 即 :

```
props:{
  msg: {
    type: String,
    default: 'hello world'
  }
},
```

需要注意的是这种传值是单向的, 你无法改变父组件的值 (当然引用类型例外) ; 而且如果直接修改props的值会报一个警告。

推荐的写法是在data()里重新定义一个变量 (见Children.vue) , 并把props赋值给它, 当然计算属性也行。

```
<template>
<section>
父组件传过来的消息是: {{myMsg}}
</section>
</template>
```

```
<script>
export default {
  name: "Children",
  components: {},
  props:['msg'],
  data() {
    return {
      myMsg:this.msg
    }
  },
  methods: {}
}
</script>
```

父组件:

```
<template>

<div class="parent">

<Children :msg="message"> </Children>

</div>

</template>

<script>

import Children from '../components/Children'

export default {

  name: 'Parent',

  components: {

    Children

  },

  data() {

    return {

      message:'hello world'

    }

  }

}
```



```
}  
  
},  
  
}  
  
</script>
```

(2) 子组件传值给父组件

这里需要使用自定义事件，在子组件中使用`this.$emit('myEvent')`触发，然后在父组件中使用`@myEvent`监听

父组件：

```
1 <template>  
2   <h1>{{cate}}栏目</h1>  
3   <ul class="list">  
4     <ArticleMainInfo v-for="article in articleArr" :ainfo="article" />  
5     <li class="end"></li>  
6     <li class="end"></li>  
7   </ul>  
8 </template>  
9  
10 <script lang="ts">  
11 import axios from "axios"; //该第三方模块已经安装过了  
12 import { defineComponent } from 'vue';  
13 //要使用的组件需要引入  
14 import ArticleMainInfo from './ArticleMainInfo.vue';  
15 // 接口  
16 interface IarticleInfoType {  
17   id: number;  
18   title: string;  
19   mainpic: string;  
20   tags: string;  
21   supports: number;  
22   comments: number;  
23   collects: number;  
24 }  
25 interface Idatatype {
```

```

26     articleArr: Array<IarticleInfoType>;
27 }
28 //defineComponent 类型推论
29 export default defineComponent({
30     data: function (): IDataType {
31         return {
32             articleArr: [], //若所需的值还未准备好，在必要时使用 null、undefined
//或者其他一些值占位
33         };
34     },
35     components: {
36         ArticleMainInfo
37     },
38     props: { /**接收的属性 */
39         cid: {
40             type: Number,
41             required: true /** 必须传值 */
42         },
43         cate: {
44             type: String,
45             required: true
46         }
47     },
48     methods: {
49         getData(cid: number) {
50             const _this = this;
51             //获取数据：发起Ajax请求
52             axios.get('/api/alist', {
53                 params: { cid }
54             }).then((res) => {
55                 //如果是箭头函数，那么this就是vue的this
56                 _this.articleArr = res.data.data;
57             }).catch(function (error) {
58                 console.log(error);
59             });
60         }
61     },
62     beforeCreate() {
63         console.log('beforeCreate...', this.$props);
64     }

```

```

65     ,
66     mounted() { /** 生命周期函数，会自动执行 */
67         console.log(this.cid, this.cate);
68         this.getData(this.cid);
69     }
70
71 });
72 </script>

```

子组件:

```

1  <template>
2      <li>
3          <div>
4              <a :href="`../detail?id=${ainfo.id}`"> </a>
5          </div>
6          <div class="tag">
7              <template v-for="tag in tags2Arr(ainfo.tags)">
8                  <a :href="`../tag?keywords=${encodeURIComponent(tag)}`">
9                      {{tag}}
10                 </a>
11                 <span>&nbsp;| &nbsp;</span>
12             </template>
13         </div>
14         <h2>
15             <a :href="`../detail?id=${ainfo.id}`">{{ainfo.title}}</a>
16         </h2>
17         <!-- 实际情况应该是进一步把上面和下面分开成2个组件 -->
18         <div class="ext">
19             <div>by Cecilia Li </div>
20             <div>{{ainfo.comments}} 评论 {{ainfo.supports}} 赞
{{ainfo.collects}} <a href="###">收藏</a></div>
21         </div>
22     </li>
23 </template>
24
25 <script lang="ts">
26 import { defineComponent } from 'vue';
27 import type { PropType } from 'vue';

```

```

28 interface IarticeMainType {
29     id: number;
30     title: string;
31     mainpic: string;
32     tags: string;
33     comments: number;
34     supports: number;
35     collects: number;
36 }
37 export default defineComponent({
38     data() {
39         return {};
40     },
41     props: {
42         ainfo: {
43             type: Object as PropType<IarticeMainType>,
44             required: true
45         }
46     },
47     methods: {
48         tags2Arr(tags: string): Array<string> {
49             return tags.split(/\s+/, 3);
50         }
51     },
52     computed: {},
53     watch: {},
54     mounted() { }
55 })
56 </script>

```

(3) 兄弟组件间传值

运用自定义事件 **emit** 的触发和监听能力，定义一个公共的事件总线 **event Bus**，通过它作为中间桥梁，我们就可以传值给任意组件了。而且通过 **event Bus** 的使用，可以加深 **emit** 的触发和监听能力，定义一个公共的事件总线 **eventBus**，通过它作为中间桥梁，我们就可以传值给任意组件了。而且通过 **eventBus** 的使用，可以加深 **emit** 的触发和监听能力，定义一个公共的事件总线 **eventBus**，通过它作为中间桥梁，我们就可以传值给任意组件了。而且通过 **eventBus** 的使用，可以加深 **emit** 的理解。

1. 一种组件间通信的方式，适用于任意组件间通信。
2. 安装全局事件总线：

```
1 new Vue({
2     .....
3     beforeCreate() {
4         Vue.prototype.$bus = this //安装全局事件总线，$bus就是当前应用的vm
5     },
6     .....
7 })
```

3. 使用事件总线：

- a. 接收数据：A组件想接收数据，则在A组件中给\$bus绑定自定义事件，事件的回调留在A组件自身。

```
1 methods(){
2     demo(data){.....}
3 }
4 .....
5 mounted() {
6     this.$bus.$on('xxxx',this.demo)
7 }
```

- b. 提供数据：this.\$bus.\$emit('xxxx',数据)

4. 最好在beforeDestroy钩子中，用\$off去解绑当前组件所用到的事件。

(4) 路由传值

i.使用问号传值

A页面跳转B页面时使用 **this.\$router.push(' /B?name=danseek')**

B页面可以使用 **this.\$route.query.name** 来获取A页面传过来的值

上面要注意 **router** 和 **route** 的区别

ii.使用冒号传值

配置如下路由：

```
{
  path: '/b/:name',
  name: 'b',
  component: () => import( '../views/B.vue')
},
```

在B页面可以通过 `this.$route.params.name` 来获取路由传入的name的值

iii.使用父子组件传值

由于router-view本身也是一个组件，所以我们也可以使用父子组件传值方式传值，然后在对应的子页面里加上props，因为type更新后没有刷新路由，所以不能直接在子页面的mounted钩子里直接获取最新type的值，而要使用watch。

```
<router-view :type="type"></router-view>
```

```
// 子页面
```

```
.....
```

```
props: ['type']
```

```
.....
```

```
watch: {
```

```
  type(){
```

```
    // console.log("在这个方法可以时刻获取最新的数据:type=",this.type)
```

```
  },
```

```
},
```

(5) 使用\$ref传值

通过\$ref的能力，给子组件定义一个ID，父组件通过这个ID可以直接访问子组件里面的方法和属性

Children.vue:

```
<template>
```

```
  <section>
```

```
    传过来的消息: {{msg}}
```

```
  </section>
```

```
</template>
```

```
<script>

export default {

  name: "Children",

  components: {},

  data() {

    return {

      msg: "",

      desc:'The use of ref'

    }

  },

  methods:{

    // 父组件可以调用这个方法传入msg

    updateMsg(msg){

      this.msg = msg

    }

  },

}

</script>
```

在父组件Parent.vue中引用Children.vue，并定义ref属性：

```
1 <template>
2   <div class="parent">
3     <!-- 给予组件设置一个ID ref="children" -->
4     <Children ref="children"></Children>
5     <div @click="pushMsg">push message</div>
6   </div>
7 </template>
8 <script>
9 import Children from '../components/Children'
```

```

10 export default {
11     name: 'parent',
12     components: {
13         Children,
14     },
15     methods: {
16         pushMsg(){
17             // 通过这个ID可以访问子组件的方法
18             this.$refs.children.updateMsg('Have you received the clothes? ')
19             // 也可以访问子组件的属性
20             console.log('children props:',this.$refs.children.desc)
21         }
22     },
23 }
24 </script>

```


(6) 使用依赖注入传给后代子孙曾孙

 假设父组件有一个方法 `getName()`，需要把它提供给所有的后代

```

provide: function () {
  return {
    getName: this.getName()
  }
}

```

 `provide` 选项允许我们指定我们想要提供给后代组件的数据/方法

然后在任何后代组件里，我们都可以使用 `inject` 来给当前实例注入父组件的数据/方法：

```
inject: ['getName']
```

 `Parent.vue`

```

1 <template>
2     <div class="parent">
3         <Children></Children>
4     </div>
5 </template>
6 <script>

```



```
7 import Children from '../components/Children'
8 export default {
9   name: 'Parent',
10  components: {
11    Children,
12  },
13  data() {
14    return {
15      name: 'dan_seek'
16    }
17  },
18  provide: function () {
19    return {
20      getName: this.name
21    }
22  },
23 }
24 </script>
25
26 Children.vue:
27 <template>
28   <section>
29     父组件传入的值: {{getName}}
30   </section>
31 </template>
32
33 <script>
34   export default {
35     name: "Children",
36     components: {},
37     data() {
38       return {
39       }
40     },
41     inject: ['getName'],
42   }
43 </script>
```

(7) 祖传孙 \$attrs

正常情况下需要借助父亲的props作为中间过渡，但是这样在父亲组件就会多了一些跟父组件业务无关的属性，耦合度高，借助\$attrs可以简化些，而且祖跟孙都无需做修改

```
1 // GrandParent.vue
2 <template>
3   <section>
4     <parent name="grandParent" sex="男" age="88" hobby="code"
      @sayKnow="sayKnow"></parent>
5   </section>
6 </template>
7 <script>
8   import Parent from './Parent'
9   export default {
10     name: "GrandParent",
11     components: {
12       Parent
13     },
14     data() {
15       return {}
16     },
17     methods: {
18       sayKnow(val){
19         console.log(val)
20       }
21     },
22     mounted() {
23     }
24   }
25 </script>
26
27 // Parent.vue
28 <template>
29   <section>
30     <p>父组件收到</p>
31     <p>祖父的名字: {{name}}</p>
32     <children v-bind="$attrs" v-on="$listeners"></children>
33   </section>
```

```

34 </template>
35 <script>
36 import Children from './Children'
37 export default {
38   name: "Parent",
39   components: {
40     Children
41   },
42   // 父组件接收了name,所以name值是不会传到子组件的
43   props: ['name'],
44   data() {
45     return {}
46   },
47   methods: {},
48   mounted() {
49   }
50 }
51 </script>
52
53 // Children.vue
54 <template>
55   <section>
56     <p>子组件收到</p>
57     <p>祖父的名字: {{name}}</p>
58     <p>祖父的性别: {{sex}}</p>
59     <p>祖父的年龄: {{age}}</p>
60     <p>祖父的爱好: {{hobby}}</p>
61
62     <button @click="sayKnow">我知道啦</button>
63   </section>
64 </template>
65
66 <script>
67 export default {
68   name: "Children",
69   components: {},
70   // 由于父组件已经接收了name属性,所以name不会传到子组件了
71   props: ['sex', 'age', 'hobby', 'name'],
72   data() {
73     return {}

```

```
74     },
75     methods: {
76         sayKnow(){
77             this.$emit('sayKnow','我知道啦')
78         }
79     },
80     mounted() {
81     }
82 }
83 </script>
```

(8)孙传祖

借助\$listeners中间事件，孙可以方便的通知祖，代码示例见7

(9)\$parent

通过parent可以获父组件实例，然后通过这个实例就可以访问父组件的属性和方法，它还有一个兄弟parent可以获父组件实例，然后通过这个实例就可以访问父组件的属性和方法，它还有一个兄弟parent可以获父组件实例，然后通过这个实例就可以访问父组件的属性和方法，它还有一个兄弟root，可以获取根组件实例。

// 获父组件的数据

this.\$parent.foo

// 写入父组件的数据

this.\$parent.foo = 2

// 访问父组件的计算属性

this.\$parent.bar

// 调用父组件的方法

this.\$parent.baz()

于是，在子组件传给父组件例子中，可以使用this.\$parent.getNum(100)传值给父组件。

(10) sessionStorage传值

sessionStorage 是浏览器的全局对象，存在它里面的数据会在页面关闭时清除。运用这个特性，我们可以在所有页面共享一份数据。

```
// 保存数据到 sessionStorage
sessionStorage.setItem('key', 'value');

// 从 sessionStorage 获取数据
let data = sessionStorage.getItem('key');

// 从 sessionStorage 删除保存的数据
sessionStorage.removeItem('key');

// 从 sessionStorage 删除所有保存的数据
sessionStorage.clear();
```

注意：里面存的是键值对，只能是字符串类型，如果要存对象的话，需要使用 `let objStr = JSON.stringify(obj)` 转成字符串然后再存储（使用的时候 `let obj = JSON.parse(objStr)` 解析为对象）。

推荐一个库 `good-storage`，它封装了 `sessionStorage`，可以直接用它的API存对象：

```
// localStorage
storage.set(key, val)
storage.get(key, def)

// sessionStorage
storage.session.set(key, val)
storage.session.get(key, val)
```

参考文章：[vue组件传值的11种方式_风中蒲公英的博客-CSDN博客_vue组件传值](#)



3通过插槽来分配内容

我们会希望能和 HTML 元素一样向组件中传递内容：

```
<AlertBox>
```

Something bad happened.

```
</AlertBox>
```

这可以通过 Vue 的自定义 **<slot>** 元素来实现：

```
<template>
```

```
<div class="alert-box">
```

```
  <strong>This is an Error for Demo Purposes</strong>
```

```
  <slot />
```

```
</div>
```

```
</template>
```

```
<style scoped>
```

```
.alert-box {
```

```
  /* ... */
```

```
}
```

```
</style>
```

如上所示，我们使用 **<slot>** 作为一个占位符，父组件传递进来的内容就会渲染在这里。



4动态组件

有些场景会需要在两个组件间来回切换，比如 Tab 界面。

通过 Vue 的 **<component>** 元素和特殊的 **is attribute** 实现的：

```
<!-- currentTab 改变时组件也改变 -->
```

```
<component :is ="currentTab"> </component>
```

在上面的例子中，被传给 **:is** 的值可以是以下几种：

- 被注册的组件名
- 导入的组件对象

你也可以使用 `is` attribute 来创建一般的 HTML 元素。

当使用 `<component :is="...">` 来在多个组件间作切换时，被切换掉的组件会被卸载。我们可以通过 `<KeepAlive>` 组件强制被切换掉的组件仍然保持“存活”的状态。

15、响应式系统原理

